A Framework for Certified Self-Stabilization

Karine Altisen, Pierre Corbineau, Stéphane Devismes

A Framework for Certified Self-Stabilization

Case Study: Silent Self-Stabilizing k-Dominating Set on a Tree

Karine Altisen Pierre Corbineau
Stéphane Devismes
VERIMAG UMR 5104, Université Grenoble Alpes, France

Abstract

We propose a general framework to build certified proofs of distributed self-stabilizing algorithms with the proof assistant Coq. We first define in Coq the *locally shared memory model with composite atomicity*, the most commonly used model in the self-stabilizing area. We then validate our framework by certifying a non trivial part of an existing silent self-stabilizing algorithm which builds a k-hop dominating set of the network. We also certified a quantitative property related to the output of this algorithm. Precisely, we show that the computed k-hop dominating set contains at most $\lfloor \frac{n-1}{k+1} \rfloor + 1$ nodes, where n is the number of nodes in the network. To obtain these results, we also developed a library which contains general tools related to potential functions and cardinality of sets.

Keywords: Self-stabilization, Proof assistant, Coq, Silent algorithms, Potential functions.

1 Introduction

In 1974, Dijkstra introduced the notion of *self-stabilizing* algorithm [Dij74] as any distributed algorithm which resumes correct behavior within finite time, regardless of the initial configuration of the system. A self-stabilizing algorithm can withstand *any* finite number of transient faults. Indeed, after transient faults hit the system and place it in some arbitrary configuration — where, for example, the values of some variables have been arbitrarily modified — a self-stabilizing algorithm is guaranteed to resume correct behavior without external (*e.g.*, human) intervention within finite time. Thus, self-stabilization makes no hypothesis on the nature or extent of transient faults that could hit the system, and recovers from the effects of those faults in a unified manner.

For more than 40 years, a vast literature on self-stabilizing algorithms has been developed. Self-stabilizing solutions have been proposed for many kinds of classical distributed problems, *e.g.*, token circulation [HC93], spanning tree construction [CYH91], clustering [CDDL10], routing [Dol97], propagation of information with feedback [BDPV99], clock synchronization [CFG92], *etc*. Moreover, self-stabilizing algorithms have been designed to handle various environments, *e.g.*, wired networks [HC93, CYH91, CDDL10, Dol97, BDPV99, CFG92], WSNs [BOBBP13, STW$^+$13], peer-to-peer systems [CDPT10, CCT13], *etc*.

Progresses in self-stabilization led to consider more and more adversarial environments. As an illustrative example, the three first algorithms proposed by Dijkstra in 1974 [Dij74] where designed for oriented ring topologies and assuming sequential executions only, while nowadays most of self-stabilizing algorithms are designed for fully asynchronous arbitrary connected networks, *e.g.*, [HC93, CDDL10, DLD$^+$12].

Consequently, the design of self-stabilizing algorithms becomes more and more intricate, and accordingly, the proofs of their respective correctness and complexity are now often tricky to establish. However,

proofs in distributed algorithmic, in particular in self-stabilization, are commonly written by hand, based on informal reasoning. This potentially lead to errors when arguments are not perfectly clear, as explained by Lamport in its position paper [Lam12]. So, in the current context, such methods are clearly pushed to their limits, since the question on confidence in proofs naturally arises. This justifies the use of a *proof assistant*, a tool which allows to develop certified proofs interactively and check them mechanically.

Contribution. In this paper, we propose a general framework to build certified proofs of self-stabilizing algorithms for wired networks with the proof assistant Coq [The12], recipient of the ACM 2013 Software system Award.

We first define in Coq the *locally shared memory model with composite atomicity*, introduced by Dijkstra [Dij74]. This model is the most commonly used in the self-stabilizing area. Our modeling is versatile, *e.g.*, it supports any class of network topologies (including arbitrary ones), the diversity of anonymity levels (from fully anonymous to fully identified), and various levels of asynchrony (*e.g.*, sequential, synchronous, fully asynchronous).

We validate our framework by certifying a non trivial part of an existing silent self-stabilizing algorithm proposed in [DLD$^+$12] which builds a k-hop dominating set of the network. Starting from an arbitrary configuration, a silent algorithm converges within finite time to a configuration from which all communication variables are constant. This class of self-stabilizing algorithms is important, as self-stabilizing algorithms building distributed data structures (such as spanning tree or clustering) often achieve the silent property, and these silent self-stabilizing data structures are widely used as basic building blocks for more complex self-stabilizing solutions, *e.g.*, [DLD$^+$12, DLD$^+$13].

Using a classical proof scheme, the certified proof consists of two main parts, one relying on partial correctness and the other on termination. For the termination part, we developed tools on potential functions and termination at a fine-grained level. Precisely, we define a potential function as a multiset containing a local potential per node. We then exploit two criteria that are sufficient to meet the conditions for using the Dershowitz–Manna well-founded ordering on multisets. Notice that the termination proof we propose for the algorithm assumes a distributed unfair daemon, the most general scheduling assumption of the model. By contrast, the proof given in [DLD$^+$12] assumes a stronger daemon, namely, a distributed weakly fair daemon.

Finally, we certify a quantitative property, as we show that the computed k-hop dominating set contains at most $\lfloor \frac{n-1}{k+1} \rfloor + 1$ nodes, where n is the number of nodes in the network. To obtain this result, we had to write a library dealing with cardinality of sets in general and properties on cardinals of finite sets *w.r.t.* basic set operations, *i.e.*, Cartesian product, disjoint union and subsets.

This work represents about 12250 lines of code (as computed by `coqwc`: 4k lines of specifications, 7k lines of proofs) written in Coq 8.4pl4 compiled with OCaml 3.11.2.

Related Work. Coq has been successfully employed for various tasks such as mathematical developments as involved in the Feit-Thompson [Ga13] theorem, formalization of the correctness of a C compiler [Ler09, MP67], certified numerical libraries [FMP13], and verification of cryptographic protocols [BABB$^+$12, CDL11].

Several works have shown that proof assistants (in particular Coq) are well-suited to certification of distributed algorithms in various contexts, *e.g.*, certification of non fault-tolerant (consequently non self-stabilizing) distributed algorithms is addressed in [CFM09, CM12, ABC$^+$13, CRTU15]. In particular, mobile distributed systems are considered in [ABC$^+$13, CRTU15]: these works are dedicated to swarms of robots that are endowed with motion actuators and visibility sensors.

Küfner *et al* [KNR12] propose to certify (using the proof assistant *Isabelle*) fault-tolerant distributed algorithms. However, the proposed framework deals with masking fault-tolerance and, consequently, is not suited to self-stabilization, which is non-masking by essence. Moreover in the modeling, the network topology is restricted to fully connected graphs.

To the best of our knowledge, only three works deal with certification of self-stabilizing algorithms [Cou02, DM09, KRS99]. A formal correctness proof of Dijkstra's seminal self-stabilizing algorithm [Dij74] is conducted with the PVS proof assistant [KRS99], however only sequential executions are considered. In [Cou02], Courtieu proposes a general setting for reasoning on self-stabilization in Coq. He restricts his study to very simple self-stabilizing algorithms (*e.g.*, the 4-states algorithm of Ghosh [Gho93]) working on networks of very restrictive topologies: *lines* and *rings*. These two works address too simple cases to draw a general framework. Finally, [DM09] proposes to certify in Coq self-stabilizing population protocols. Population protocols are used as a theoretical model for a collection (or population) of tiny mobile agents that interact with one another to carry out a computation. The movement pattern of the agents is unpredictable, but subject to some fairness constraints, and computations must eventually converge to the correct output value in any schedule that results from that movement. In such a model, communication is implicit and there is no notion of communication network: all pairs of agents interact infinitely often. Hence, this latter work is not relevant for wired networks, as considered here.

Roadmap. The rest of the paper is organized as follows. In the next section, we describe how we define the locally shared memory model with composite atomicity in Coq. In Section 3, we express the definitions of self-stabilization and silence in Coq, moreover we give a sufficient condition to show that an algorithm is silent and self-stabilizing. In Section 4, we present our case study. The three next sections are dedicated to the proof in Coq of the case study: partial correctness (Section5), termination (Section 6), and size of computed k-hop dominating set (Section 7). We make concluding remarks and perspectives in Section 8.

In this report, we present the work together with few pieces of Coq code that we simplify in order to make them readable. In particular, we intend to use notations, as defined in the model or in the algorithm, in those pieces of code. The Coq definitions, lemmas, theorems, and documentation related to this report are available as an online browsing at `http://www-verimag.imag.fr/~altisen/PADEC/`. All source codes are also available at this address. We encourage the reader to visit this web page for a deeper understanding of our work.

2 Locally Shared Memory Model in Coq

In this section, we explain how we model in Coq the *locally shared memory model with composite atomicity*. This model has been introduced by Dijkstra [Dij74], and since then is the most commonly used in the self-stabilizing area.

2.1 Distributed Systems

We define a *distributed system* as a finite set of interconnected nodes. Each node has its own private memory and runs its own code. It can also interact with other nodes in the network *via* interconnections. The model in Coq reflects this by defining two independent classes:

- A `Network` is equipped with a type `Node`, representing nodes of the network. A `Network` defines functions and properties that depict its topology, *i.e.*, interconnections between nodes. Those interconnections are specified using the type `Channel`.

- An `Algorithm` is equipped with a type `State` which describes the memory state of nodes. Its main function is `run` which specifies how the codes of nodes execute and interact using channels (type `Channel`).

2.2 Network and Topology

Nodes in a distributed system can directly communicate with a subset of other nodes. As commonly done in the literature, we view the communication *network* as a simple directed graph $G = (V, E)$, where V is set of vertices representing nodes and $E \subseteq V \times V$ is a set of edges representing direct communication between distinct nodes. We note $n = |V|$ the numbers of nodes.

Two distinct nodes p and q are said to be *neighbors* if $(p, q) \in E$. From a computational point of view, p uses a distinct channel $c_{p,q}$ to communicate with each of its neighbors q: it does not have direct access to q. In the type `Network`, the topology is defined using this narrow point of view, *i.e.*, interconnections (edges of the graph) are represented using channels only. In particular, the neighborhood of p is encoded with the set $\mathcal{N}_p$ which contains all channels $c_{p,q}$ outgoing from p. The sets $\mathcal{N}_p$, for all p, are modeled in Coq as lists, using the function `(peers: Node → list Channel)`. The function `(peer: Node → Channel → option Node)` returns the destination neighbor for a given channel name, *i.e.*, `(peer` p $c_{p,q}$`)` returns `(Some` q`)`, or $\perp$[1] if the name is unused. We also define the shortcut ternary relation `(is_channel p c p')` as `(peer p c)` equals `(Some p')` where `p` and `p'` are nodes and `c` a channel.

Communications can be made bidirectional, assuming a property called `sym_net`, which states that for all nodes p_1 and p_2, the network defines a channel from p_1 to p_2 if and only if it also defines a channel from p_2 to p_1. In case of bidirectional links (p, q) and (q, p) in E, p can access its channel name at q using the function ρ_p. Thus, we have the following identities: $\rho_p(c_{p,q})$ equals $c_{q,p} \in \mathcal{N}_q$ and $\rho_q(c_{q,p})$ equals $c_{p,q} \in \mathcal{N}_p$. In Coq, the role of ρ_p is assigned to the function `(reply_to: Node → Channel → Channel)`.

As last requirement, we suppose that, since the number of nodes in the network is finite, we have a list, called `all_nodes`, containing all the nodes. In particular, this assumption makes the *emptiness test* decidable: this test states that for any function `(f: Node → option A)` (with `A`, some type), one can compute whether `f` always returns $\perp$ for any parameter. This test is used in the framework to detect termination of the algorithm.

As a means of checking actual usability of the `Network` type definition, we have defined a function that can build any finite `Network` from a description of its topology given by a list of lists of neighbors.

2.3 Computational Model

In the *locally shared memory model with composite atomicity*, nodes communicate with their neighbors using finite sets of locally shared registers, called *variables*. A node can read its own variables and those of its neighbors, but can only write to its own variables. Each node operates according to its local *program*.

Distributed Algorithm. A *distributed algorithm* $\mathcal{A}$ is defined as a collection of n *programs*, each operating on a single node. The *state* of a node in $\mathcal{A}$ is defined by the values of its local variables and is represented using an abstract immutable Coq datatype `State`. Such a datatype is usually implemented as a record containing the values of the algorithm variables. A node p can access the states of its neighbors

[1] Option type is used for partial functions which, by convention, returns `(Some _)` when defined, and `None` otherwise. `None` is denoted by $\perp$ in this paper.

using the corresponding channels: we call this the *local configuration* of p, and model it as a function typed `(Local_Env := Channel → option State)` which returns the current state of a neighbor, given the name of the corresponding channel (or $\perp$ for an invalid name).

The program of each node p in $\mathcal{A}$ consists of a finite set of guarded actions:

$$\langle guard \rangle \hookrightarrow \langle statement \rangle$$

The *guard* is a Boolean expression involving variables of p and its neighbors. The *statement* updates some variables of p. An action can be executed only if its guard evaluates to $true$; in this case, the action is said to be *enabled*. A node is said to be *enabled* if at least one of its actions is enabled. The local program at node p is modeled by a function `run` of type `(list Channel →(Channel → Channel) → State → Local_Env →option State)`. This function is given access to the local topology and states around p. It takes as first two arguments $\mathcal{N}_p$ and ρ_p. It then takes as inputs the current state of p and its current local configuration. The returned value is the next state of node p if p is enabled, $\perp$ otherwise. Note that `run` provides a functional view of the algorithm: it includes the whole set of possible actions, but returns a single result; this model is thus restricted to *deterministic algorithms*.[2]

Semantics. A *configuration* `g` of the system is defined as an instance of the states of all nodes in the system, *i.e.*, a function typed `(Env := Node → State)`. For a given node `p` and configuration `(g: Env)`, the term `(g p)` represents the state of `p` in configuration `g`. Thanks to this encoding, we easily obtain the local configuration (type `Local_Env`) of node `p` by composing `g` and `peer` as a function `(local_env g p) := (fun (c: Channel) => option_map g (peer p c))` which returns `(g p')` when `(peer p c)` returns `Some p'`, and $\perp$ otherwise. Hence, the execution of the algorithm on node `p` in current configuration `g` is obtained by: `(run` $\mathcal{N}_{\mathtt{p}}$ $\rho_{\mathtt{p}}$ `(g p) (local_env g p))`; it returns either $\perp$ if the node is disabled or `(Some s)` where `(s: State)` is next state of `p`. We define `(enabled_b g p)` as the Boolean value (type `bool`) which returns `true` if node `p` is enabled in configuration `g` and `false` otherwise.

Assume the system is in some configuration `g`. If there exist some enabled nodes, a *daemon*[3] selects a non-empty set of them; every chosen node *atomically* executes its algorithm, leading to a new configuration `g'`. The transition from `g` to `g'` is called a *step*. To model steps in Coq, we use functions with type `(Diff := Node → option State)`. We simply call *difference* a variable `d` of type `Diff`. A difference contains the updated states of the nodes that actually execute some action during the step, and maps any other node to $\perp$. We define the predicate `valid_diff` that qualifies the current configuration and a difference expressing the result of a step. It holds when at least one node actually moves and every update in the difference corresponds to the execution of the algorithm, namely, `run`. Next configuration, `g'`, is then obtained applying the function `(diff_eval d g)` such that: $\forall$`p, (g' p) = (d p)` if `(d p)` $\neq \perp$, and `(g' p) = (g p)` otherwise.

Steps induce a binary relation $\mapsto$ over configurations defined in Coq by the relation `Step`: `(Step g' g)` expresses that `g` $\mapsto$ `g' (Step g' g)`[4], meaning that `g` $\mapsto$ `g'` is actually a valid step, *i.e.*, there exists some valid difference `d` for `g` (`valid_diff g d`) and `g'` is equal to `(diff_eval d g)`. An *execution* of $\mathcal{A}$ is a sequence of configurations $\mathtt{g}_0\ \mathtt{g}_1\ \ldots\ \mathtt{g}_i\ \ldots$ such that $\mathtt{g}_{i-1} \mapsto \mathtt{g}_i$ for all $i > 0$. Executions may be finite or infinite and are modeled in Coq with the type

```
CoInductive Exec: Type :=
  | e_one:  Env → Exec
```

[2]Finite non-determinism could be handled by having `run` output `list State` instead of `option State`.

[3]The daemon achieves the asynchrony of the system.

[4]Please, note the inverse order of the parameters in `Step`.

```
  | e_cons: Env → Exec → Exec.
```

and the predicate

```
CoInductive valid_exec: Exec → Prop :=
  | v_one:  ∀ (g: Env), valid_exec (e_one g)
  | v_cons: ∀ (e: Exec) (g: Env),
              valid_exec e → Step (Fst e) g → valid_exec (e_cons g e).
```

where `(Fst e)` returns the first configuration of execution `e`. The keyword **CoInductive** generates a greatest fixed point capturing potentially infinite constructions[5]. A variable `(e: Exec)` actually represents an (valid) execution of $\mathcal{A}$ when `(valid_exec e)` holds, since each pair of consecutive configurations `g`, `g'` in `e` satisfies `(Step g' g)`.

Maximal executions are either infinite, or end at a *terminal* configuration in which no action of $\mathcal{A}$ is enabled at any node. Terminal configurations are detected in Coq using the proposition `(terminal g)`, for a configuration `g`, which holds when *every node* computes `run` from `g` and returns $\perp$. Note that this predicate is decidable thanks to the emptiness test. A maximal execution is described by the coinductive proposition:

```
CoInductive max_exec: Exec → Prop :=
| max_one: ∀ (g: Env), terminal g → max_exec (e_one g)
| max_cons: ∀ (g: Env) (e: Exec), max_exec e → max_exec (e_cons g e).
```

As previously stated, each step from a configuration to another is driven by a *daemon*. In our case study, we assume that the daemon is *distributed* and *unfair*. *Distributed* means that while the configuration is not terminal, the daemon should select at least one enabled node, maybe more. *Unfair* means that there is no fairness constraint, *i.e.*, the daemon might never select an enabled node unless it is the only one enabled. Notice that the propositions `valid_diff`, `Step` and henceforth `valid_exec` are sufficient to handle the distributed unfair daemon.

Types are Setoids. When using Coq function types to represent configurations and differences, we need to state pointwise function equality, which equates to functions having equal values (extensional equality). The Coq default equality is inadequate for functions since it asserts equality of implementations (intensional equality). So, instead we chose to use the setoid paradigm: we endow every base type with an *equivalence relation*. Setoids are commonly used in Coq for subsets, function sets, and to represent set-theoretic quotient sets (such as rational numbers or real numbers); in particular we make use of libraries `Coq.Setoids.Setoid` and `Coq.Lists.SetoidList`. Furthermore, we assume those equivalence relations are *decidable*.

For instance, the equality for type `Node` is noted `(eqN: relation Node)` and assumes

```
eqN_equiv: Equivalence eqN
eqN_dec: Decider eqN
```

Note that `(relation Node)` stands for `(Node → Node →` **Prop**`)`; `Equivalence` defines the conjunction of reflexivity, symmetry, and transitivity of the relation; `(Decider eqN)` expresses that the relation is decidable by `(∀(p p': Node), {eqN p p'} + {¬ eqN p p'})`, where `{A} + {B}` is the standard Coq notation for computational disjunction between `A` and `B`, *i.e.*, Booleans carrying proofs of `A` or `B`.

[5] As opposed to this, the keyword **Inductive** only captures finite constructions.

Compatibility. Consequently, every function type is endowed with an equality *partial equivalence relation* (*i.e.*, symmetric and transitive) which states that, given equivalent inputs, the outputs of two equivalent functions are equivalent. For instance, the equality for the type `Env` is defined by

```
eqE := (eqN ==> eqS)
```

(where `eqS` is the decidable equivalence relation on type `State`) and means that for any configuration `g1` and `g2`, `(eqE g1 g2)` expresses that

```
∀(p1 p2: Node), eqN p1 p2 -> eqS (g1 p1) (g2 p2)
```

(in this model, `(eqN p1 p2)` means that `p1` and `p2` represent the same node in the network). Note that `eqE` is not reflexive *a priori*. However, we can only reason about functions equivalent to themselves: those functions are called *compatible*, *i.e.*, they return equivalent results when executed with equivalent parameters. Back to the example of configurations, we will require that any configuration `(g: Env)` is compatible, namely that

```
Proper EqE g
```

This means that for two equivalent nodes `p1` and `p2`, *i.e.*, such that `(eqN p1 p2)`, we expect that `(g p1)` and `(g p2)` produce the same result with respect to `eqS`: `(eqS (g p1) (g p2))`.

In all the framework, we assume *compatible configurations only*. Also, we require that `run` is compatible, meaning that it returns equivalent results when executed with equivalent parameters (*e.g.*, a permutation on the list of channels $\mathcal{N}_p$ or a compatible function equal to `(local_env g p)`).

Read-Only Variables. We allow a part of a node state to be read-only: this is modeled with the type `ROState` and by the function `(RO_part: State → ROState)` which typically represents a subset of the variables handled in the `State` of the node. The projection `RO_part` is extended to configurations by the function `(ROEnv_part g := (fun (p: Node) => RO_part (g p)))`, which returns a value of type `ROEnv`. We add the property `RO_stable` to express the fact that those variables are actually read-only, namely no execution of `run` can change their values. From the assumption `RO_stable`, we show that any property defined on the read-only variables of a configuration is indeed preserved during steps.

The introduction of Read-Only variables has been motivated by the fact that we want to encompass the diversity of anonymity levels from the distributing computing literature, *e.g.*, fully anonymous, semi-anonymous, rooted, fully identified networks, *etc*. By default (with empty `RO_part`), our Coq model defines fully anonymous network thanks to the distinction between nodes (type `Node`) and channels (type `Channel`). We enriched our model to reflect other assumptions.

For example, consider the fully identified assumption. Identifiers are typically constant integers, stored in the node states. In our model, they would be stored in the read-only part of the state. Furthermore, identifiers should be constant and unique all along the execution of the algorithm. This means they should be unique in initial configuration and kept constant during the whole execution.

We define a predicate `Assume_RO` on `ROEnv` (in the case of fully identified assumption, `Assume_RO` would express uniqueness of identifiers) that will be assumed at each initial configuration. From `RO_stable`, this property will remain true all along any execution. Furthermore, the predicate `Assume_RO` can express other assumptions on the network such as connected networks or tree networks (for this latter, see the case study). As a shortcut, for any configuration `(g: Env)`, we use notation `Assume g := (Assume_RO (ROEnv_part g))`.

3 Self-Stabilization and Silence

In this section, we express self-stabilization [Dij74] in the locally shared memory model with composite atomicity using Coq properties.

Self-Stabilization. Consider a distributed algorithm $\mathcal{A}$. Let $\mathbb{S}$ be a predicate on executions (type `(Exec → Prop)`). $\mathcal{A}$ is *self-stabilizing w.r.t. specification* $\mathbb{S}$ if there exists a predicate $\mathbb{P}$ on configurations (type `(Env → Prop)`) such that:

- $\mathbb{P}$ is *closed* under $\mathcal{A}$, *i.e.*, for each possible step `g ↦ g'`, `(ℙ g)` implies `(ℙ g')`:

```
closure ℙ := ∀ (g g': Env), Assume g → ℙ g → Step g' g → ℙ g';
```

- $\mathcal{A}$ *converges* to $\mathbb{P}$, *i.e.*, every execution of $\mathcal{A}$ contains a configuration which satisfies $\mathbb{P}$:

```
convergence ℙ := ∀ (e: Exec),
  Assume (Fst e) → valid_exec e → max_exec e →
  safe_suffix (fun suf: Exec => ℙ (Fst suf)) e,
```

  where `(safe_suffix S e)` inductively checks that execution `e` contains a suffix that satisfies `S`.

- $\mathcal{A}$ *meets* $\mathbb{S}$ from $\mathbb{P}$, *i.e.*, every maximal execution, from configurations which satisfy $\mathbb{P}$, satisfies $\mathbb{S}$:

```
spec_ok ℙ 𝕊 := ∀ (e: Exec),
  Assume (Fst e) → valid_exec e → max_exec e → ℙ (Fst e) → 𝕊 e.
```

The configurations which satisfy the predicate $\mathbb{P}$ are called *legitimate configurations*. The following predicate charaterizes a self-stabilizing algorithm:

```
self_stab 𝕊 := ∃ℙ, closure ℙ ∧ convergence ℙ ∧ spec_ok 𝕊 ℙ.
```

Silence. An algorithm is *silent* if the communication between the nodes is fixed from some point of the execution [DGS96]. This latter definition can be transposed in the locally shared memory model as follows: $\mathcal{A}$ is *silent* if all its executions are finite.

```
Inductive finite_exec: Exec → Prop :=
  | f_one: ∀ (g: Env), finite_exec (e_one g)
  | f_cons: ∀ (e: Exec) (g: Env),
               finite_exec e → finite_exec (e_cons g e)

silence := ∀ (e: Exec), Assume (Fst e) → valid_exec e → finite_exec e
```

By definition, executions of a *silent and self-stabilizing* algorithm *w.r.t* some specification $\mathbb{S}$ end in configurations which are usually used as legitimate configurations, *i.e.*, satisfying $\mathbb{P}$. In this case, $\mathbb{S}$ can only allow constrained executions made of a single configuration which is legitimate; $\mathbb{S}$ is then noted $\mathbb{S}_{\mathbb{P}}$. To prove that $\mathcal{A}$ is both silent and self-stabilizing *w.r.t.* $\mathbb{S}_{\mathbb{P}}$, we use, as commonly done, a sufficient condition which requires to prove that

- all terminal configurations of $\mathcal{A}$ satisfy $\mathbb{P}$:

```
P_correctness ℙ :=
  ∀ (g: Env), Assume g → terminal g → SPEC g
```

- and all executions of $\mathcal{A}$ are finite:

```
termination := ∀ (g: Env), Assume g → Acc Step g.
```

The latter property is expressed with `(Acc Step g)` for every configuration `g`. The inductive proposition `Acc` is taken from Library `Coq.Init.Wf` which provides tools on well-founded induction. The accessibility predicate `(Acc Step g)` is translated into

```
(∀ g': Env, Step g' g → Acc Step g') → Acc Step g
```

Namely, the base case of induction holds when no step is possible from current configuration `g` and then, inductively, any configuration `g'` that can reach such a terminal configuration satisfies `(Acc Step g')`.

The sufficient condition, used to prove that an algorithm is both silent and self-stabilizing, is expressed and proved by:

Lemma `silent_self_stab (ℙ: Env → Prop):`
`P_correctness ℙ ∧ termination → silence ∧ self_stab 𝕊ℙ.`

4 Case Study

To validate our framework, we certified a non trivial part of an existing silent self-stabilizing algorithm proposed in [DLD+12]. Given a non-negative integer k, this algorithm builds a k-clustering of a bidirectional connected network $G = (V, E)$ which contains at most $\lfloor \frac{n-1}{k+1} \rfloor + 1$ k-clusters, where n is the number of nodes. A *k-cluster* of G is defined to be a set $C \subseteq V$, together with a designated node *Clusterhead*$(C) \in C$, such that each member of C is within distance k of *Clusterhead*(C), where the distance between any two nodes p and q (noted $\|p, q\|$ in the following) is the length of a shortest path linking p to q in G. A *k-clustering* of G is a partition of V into distinct k-clusters. The k-clustering problem is strongly related to the notion of *k-hop dominating set*. A subset of nodes D is a k-hop dominating set of G if every node is within distance k from some member of D. So, by definition, the set of clusterheads of any k-clustering is a k-hop dominating set.

The algorithm proposed in [DLD+12] is actually a hierarchical collateral composition [DLD+13] of two silent self-stabilizing sub-algorithms: the former builds a particular kind of rooted spanning tree, called an MIS *tree*;[6] the latter is a k-clustering construction which stabilizes once a rooted spanning tree is available in the network.

The crucial part of the second sub-algorithm consists in computing (in a self-stabilizing and silent way) a k-hop dominating set D of size at most $\lfloor \frac{n-1}{k+1} \rfloor + 1$ in an arbitrary rooted spanning tree. D will designate the set of clusterheads in the computed k-clustering. This task is performed using the 1-rule Algorithm $\mathcal{D}(k)$, whose code is given in Algorithm 1.

We have used our framework to build a certified proof which shows that $\mathcal{D}(k)$ is a silent and self-stabilizing algorithm for building a k-hop dominating set of at most $\lfloor \frac{n-1}{k+1} \rfloor + 1$ nodes in any network equipped with a rooted spanning tree.

The first step has consisted in encoding $\mathcal{D}(k)$ in our framework. We translated the unique rule of the algorithm into the type `Algorithm`. Moreover, we expressed the assumptions on the network G in the predicate `Assume`: G is bidirectional and a rooted spanning tree is available in G, *n.b.*, this latter also implies that G is connected. In the sequel, the spanning tree and its root are respectively denoted by $T = (V, E')$ and $r \in V$. Moreover, for any node p, $T(p)$ denotes the subtree of T with root p.

We certified the correctness part and then the termination part of the specification. For the latter property, the proof we propose assumes a *distributed unfair daemon*. By contrast, the proof given in [DLD+12] assumes a stronger daemon (namely, a weakly fair distributed daemon). Finally, we provided a certified

[6] An MIS tree is a spanning tree whose nodes at even levels form a maximal independent set (MIS) of the network.

Algorithm 1 $\mathcal{D}(k)$, code for each process p

Constant Input: $\texttt{Par}(p) \in \mathcal{N}_p \cup \{\perp\}$

Variable: $p.\alpha \in \{0, ..., 2k\}$

Predicates:

$$\begin{array}{lcl} \textit{IsRoot}(p) & \equiv & \texttt{Par}(p) = \perp \\ \textit{IsShort}(p) & \equiv & p.\alpha < k \\ \textit{IsTall}(p) & \equiv & p.\alpha \geq k \end{array}$$

$$\textit{kDominator}(p) \equiv (p.\alpha = k) \vee (\textit{IsShort}(p) \wedge \textit{IsRoot}(p))$$

Macros:

$$\begin{array}{lcl} \textit{Children}(p) & = & \{q \in \mathcal{N}_p \mid \texttt{Par}(q) = \rho_p(q)\} \\ \textit{ShortChildren}(p) & = & \{q \in \textit{Children}(p) \mid \textit{IsShort}(q)\} \\ \textit{TallChildren}(p) & = & \{q \in \textit{Children}(p) \mid \textit{IsTall}(q)\} \end{array}$$

$$\begin{array}{lcl} \textit{MaxAShort}(p) & = & \textbf{if } \textit{ShortChildren}(p) = \emptyset \textbf{ then } -1 \\ & & \textbf{else } \max\{q.\alpha \mid q \in \textit{ShortChildren}(p)\} \end{array}$$

$$\begin{array}{lcl} \textit{MinATall}(p) & = & \textbf{if } \textit{TallChildren}(p) = \emptyset \textbf{ then } 2k+1 \\ & & \textbf{else } \min\{q.\alpha \mid q \in \textit{TallChildren}(p)\} \end{array}$$

$$\begin{array}{lcl} \textit{Alpha}(p) & = & \textbf{if } \textit{MaxAShort}(p) + \textit{MinATall}(p) \leq 2k-2 \textbf{ then } \textit{MinATall}(p)+1 \\ & & \textbf{else } \textit{MaxAShort}(p)+1 \end{array}$$

Action:

$$p.\alpha \neq \textit{Alpha}(p) \;\hookrightarrow\; p.\alpha \leftarrow \textit{Alpha}(p)$$

proof on the maximal size of the computed k-hop dominating set. The design of the proofs led us to develop general tools in our framework. In particular, we proposed general theorems to prove termination and tools for counting elements in sets.

4.1 Algorithm $\mathcal{D}(k)$

Local States. In the algorithm, the knowledge of T is locally distributed at each node p using the constant input $\texttt{Par}(p) \in \mathcal{N}_p \cup \{\perp\}$. When $p \neq r$, $\texttt{Par}(p) \in \mathcal{N}_p$ and designates its parent in the tree. Otherwise, p is the root and $\texttt{Par}(p) = \perp$. Then, each node p maintains a single variable: $p.\alpha$, an integer in range $\{0, ..., 2k\}$. We have instantiated the Coq `State` of a node as a record containing fields `(Par: option Channel)` and `(α: Z)`. `(Par p)` stands for $\texttt{Par}(p)$ and is the unique *read-only* variable for `p`. Moreover, `(α p)` stands for $p.\alpha$ and is taken in `Z` (integer). Indeed, we choose to encode every number in the algorithm as integer in `Z`, since some of them may be negative (see *MaxAShort*) and a computation uses minus (see *Alpha*). Furthermore, we have proven $p.\alpha$ is in range $\{0, ..., 2k\}$ after p participates in any step and also when the system is in a terminal configuration.

Spanning Tree Assumption. We use the predicate `(span_tree r Par)` on the read-only variables `Par` to express the assumption on the spanning tree. First, by construction, each node `p` has exactly at most one parent `(Par p)`. Then, `(span_tree r Par)` checks that the graph T induced by `Par` is a subgraph of G which actually encodes a spanning tree rooted at *r* by the conjunction of the following predicates.

- Predicate `parent_is_peer` means that for every non-root node `p`, any link from `p` to `(Par p)` is an existing channel outgoing from `p` (*i.e.*, is in $\mathcal{N}_\texttt{p}$):

  ```
  parent_is_peer := ∀(p: Node) (c: Channel),
    eqoptionA eqC (Par p) (Some c) → ∃(q: Node), is_channel p c q
  ```

  where `eqC` denotes equality between channels and `eqoptionA eqC` checks whether parameters are either both $\perp$, or `(Some c1)` and `(Some c2)` with `(eqC c1 c2)`.

- Predicates `is_root_root` and `is_root_unique` say that *r* is the unique node with no parent, *i.e.*, such that `(Par r)` is $\perp$:

  ```
  is_root_root := is_root r
  is_root_unique := ∀(p1 p2: Node), is_root p1 → is_root p2 → eqN p1 p2
  ```

  where predicate `(is_root p)` checks whether `(Par p)` equals $\perp$.

- Finally, predicate `no_loop` requires that there is no loop in T, following `Par` pointers, namely, the transitive closure of `Par` is not reflexive:

  ```
  no_loop: ∀(p: Node), ¬transitive_closure (is_tree_parent_of Par) p p
  ```

  where `is_tree_parent_of` is the relation between nodes and their parents in T, extracted from `Par` and `(transitive_closure R)` is the relation obtained from the transitive closure of relation `R`.

From the last point, we show that, since the number of nodes is finite, the relation extracted from `Par` between nodes and their parents (resp. children) in T is well-founded:

```
WF_par := well_founded (is_tree_parent_of)
WF_child := well_founded (flip (is_tree_parent_of))
```

where `(flip is_tree_parent_of)` is the relation between nodes and their children (since `flip` provides the inverse relation); and `(well_founded R := ∀a, Acc R a)` means that relation `R` is well-founded (primitive `well_founded` is taken from standard Coq Library `Coq.Init.Wf`, as `Acc`).

Finally, we express that a spanning tree exists, rooted at *r* and using `Par` by the predicate `(span_tree r Par)` which is the conjunction of the four predicates above (namely `is_root_unique`, `is_root_root`, `parent_is_peer` and `no_loop`). We now instantiate predicate `Assume`, for any configuration `(g: Env)` with read-only part noted `Par` as:

```
Assume_kdom g := sym_net ∧ ∃ (r: Node), span_tree r Par.
```

k-hop Dominating Set. The goal of $\mathcal{D}(k)$ is to compute an output predicate *kDominator*(p) for every node p (see Algorithm 1 for its definition) in such way that the system converges to a terminal configuration in which the set $Dom = \{p \in V \mid \textit{kDominator}(p)\}$ defines a k-hop dominating set of T (and so of G). We pose an arbitrary positive parameter k, taken in `Z` as for other numbers, with assumption that it is positive. We define the expected specification using predicate $\mathbb{P}_{\texttt{kDom}}$ on configurations, where $\mathbb{P}_{\texttt{kDom}}$ holds in configuration `g` if and only if the set $Dom = \{p \in V \mid \textit{kDominator}(p)\}$ is a k-hop dominating set of T:

```
P_kDom g := ∀(p: Node), ∃(kdom: Node), (kDominator g kdom) ∧
  ∃(path: list Node), (is_path g path kdom p) ∧ (length path) ≤ k.
```

where the predicate `is_path` detects if the list of nodes `path` actually represents a path on the tree T between the nodes `kdom` and `p`, and `length` computes the length of the path.

$\mathcal{D}(k)$ in Coq. Every predicate and macro of Algorithm 1 has been directly encoded in Coq. For a node `p` and current configuration `g`, all predicates and functions depend on $\mathcal{N}_{\texttt{p}}$, $\rho_{\texttt{p}}$, `(g p)` and `(local_env g p)`. The translation is quasi-syntactic as shown in the following two examples:[7]

```
kDominator p := orb (Z.eqb p.(alpha) k) (andb (IsShort p) (IsRoot p))

Alpha p := if Z_le_gt_dec (MaxAShort p + MinATall p + 2) (2 * k)
           then MinATall p + 1
           else MaxAShort p + 1

run p := if Z.eq_dec (α p) (Alpha p)
         then ⊥
         else Some {| α := Alpha p; Par := (Par p) |}
```

(`Z.eqb` stands for Boolean equality (return type `bool`) between integer numbers in `Z`; `orb` (resp. `andb`) stands for Boolean-or (resp. -and). `Z_le_gt_dec` is a decidable comparator $\leq$ between integer numbers of type `Z` and `Z.eq_dec` is a decidable equality.) Remind that the fact that node p is not enabled at current step is encoded by $\bot$. Finally, the definition of $\mathcal{D}(k)$, of type `Algorithm`, comes with a proof that `run` is compatible, as composition of compatible functions, and also with a straightforward proof of `RO_stable` which asserts that the read-only part of the state, `Par`, is constant during steps, when applying `run`.

[7] Notice that `p` should be instantiated by `(g p)` in those definitions; we adopt this writing to make Coq definitions resemble Algorithm 1.

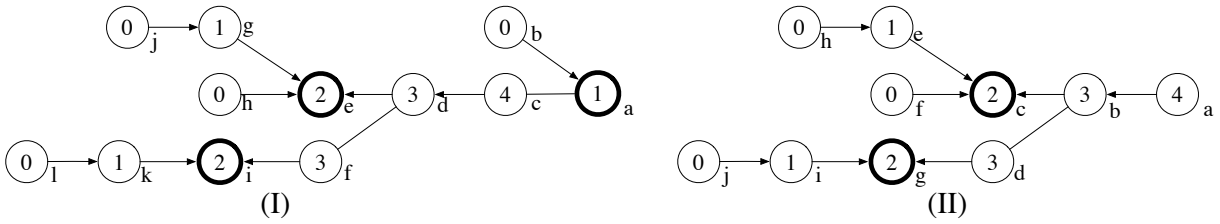

Figure 1: Two examples of 2-hop dominating sets computed by $\mathcal{D}(2)$. We only draw the spanning tree, other edges are omitted. The root of each tree is the rightmost node. α-values are given inside the nodes. Bold circles represent members of Dom. Arrows represent the path from nodes to their associated witnesses.

Overview of $\mathcal{D}(k)$. Algorithm $\mathcal{D}(k)$ computes a k-hop dominating set of T, noted Dom, using the variable α at each node. Precisely, Dom is defined as the set of nodes p such that *kDominator*(p) holds, *i.e.*, where $p.\alpha = k$, or $p.\alpha < k$ and $p = r$. Dom is constructed by dynamic programming, in a bottom-up fashion starting from the leaves of T. The goal of variable $p.\alpha$ at each node p is twofold:

- Variable $p.\alpha$ allows to determine a path of length at most k from p to some node q of Dom which acts as a *witness* for guaranteeing the k-hop domination of Dom. Consequently, q will be denoted as $Witness(p)$ in the following.
- Once correctly evaluated, the value $p.\alpha$ is equal to $\|p, q\|$, where q is the furthest node in $T(p)$ that has the same witness as p.

We divide processes into *short* and *tall* according to the value of their α-variable: If p satisfies *IsShort*(p), *i.e.*, $p.\alpha < k$, then p is said to be *short*; otherwise, p satisfies *IsTall*(p) and is said to be *tall*. In a terminal configuration, the meaning of $p.\alpha$ depends on whether p is *short* or *tall*.

(i) If p is *short*, we have two cases: $p \neq r$ or $p = r$. In the former case, $Witness(p) \in Dom$ is outside of $T(p)$, that is, the path from p to $Witness(p)$ goes through the parent link of p in the tree, and the distance from p to $Witness(p)$ is at most $k - p.\alpha$. See, for example, in Configuration (I) of Figure 1, $k = 2$ and $g.\alpha = 0$ mean that $Witness(g)$ is at most at distance $k - 0 = 2$, now its witness d is at distance 1. In the latter case, p is not k-hop dominated by any other process of Dom inside its subtree and, by definition, there is no process outside its subtree, indeed $T(p) = T$, see the root a in Configuration (I) of Figure 1. Thus, p must be placed in Dom.

(ii) If p is *tall*, there is a process q at $p.\alpha - k$ hops below p such that $q.\alpha = k$. So, $q \in Dom$ and p is k-hop dominated by q. Hence, $Witness(p) = q$. The path from p to $Witness(p)$ goes through its tall child with minimum α-value. See, for example, in Configuration (I) of Figure 1, $k = 2$ and $c.\alpha = 3$ mean that $Witness(c)$, here d, is $3 - k = 1$ hop below c. Note that, if $p.\alpha = k$, then $p.\alpha - k = 0$, that is, $p = q = Witness(p)$ and p belongs to Dom.

In $\mathcal{D}(k)$, $p.\alpha$ is computed using macro *Alpha*(p) (see Algorithm 1). Two examples of 2-clustering computed by $\mathcal{D}(2)$ are given in Figure 1. In Subfigure 1.(I), the root is a *short* process, consequently it belongs to Dom. In Subfigure 1.(II), the root is a *tall* process, consequently it does not belong to Dom.

Detailed Explanation of $\mathcal{D}(k)$. According to the macro *Alpha*(p), $p.\alpha$ is computed in a bottom-up fashion in T using the unique action of the algorithm.

Consider a *leaf* f. By definition, *MaxAShort*(f) + *MinATall*$(f) = -1 + 2k + 1 > 2k - 2$. Thus, $f.\alpha = -1 + 1 = 0$, which corresponds to the distance between f and its furthest descendant that will have

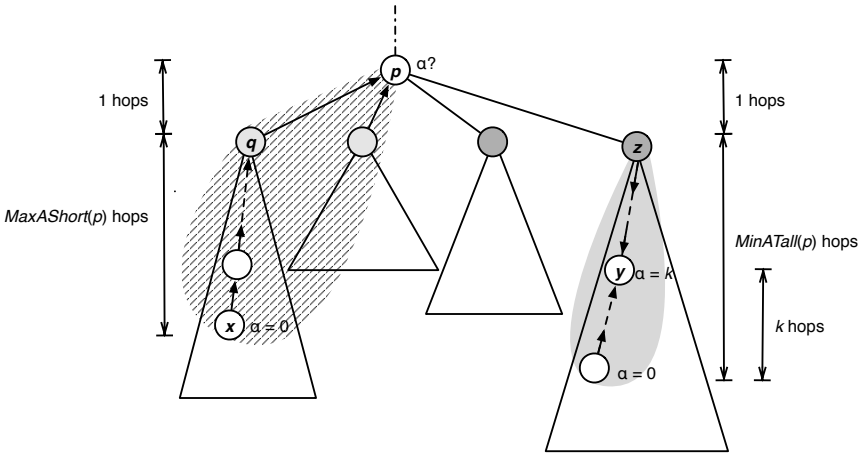

Figure 2: Illustrative example: Light gray nodes are *short* children of p; gray nodes are *tall* children of p. The shade area shows the nodes that already choose the same witness as p. The light gray area shows the nodes that already choose the same witness as z.

the same witness (actually f itself).

Consider now any *internal node* p and assume that the α-variables of all its children are correctly evaluated. p should choose a witness that will be either (1) in its subtree (in this case, p will be *tall*), or (2) outside its subtree (in this case, p will be *short*). We should preferably make the choice (1) to reduce the number of witnesses, and so the size of Dom.

Let q be a *short* child of p. From bullet (i) in the previous paragraph, the path from q to its witness goes through p. Thus, to prevent cycle creation,

$(*)$ p must not choose a witness which is in the subtree of any of its short children.

From now on, follow the illustrative example given in Figure 2. Let x be the furthest node that is in the subtree of some *short* child of p and has the same witness as p. Let q be the *short* child of p such that $x \in T(q)$. Then, from bullet (i) in the previous paragraph, x is at distance *MaxAShort*$(p) + 1$ from p. Two cases are then possible:

- Assume *MaxAShort*(p) + *MinATall*$(p) > 2k - 2$. If p chooses a node y of its subtree as witness, then from $(*)$, the path from p to y should go through one of its *tall* children. So, p will be at least at distance *MinATall*$(p) - k + 1$ from y, by (ii). Now, in this case, x will be at least at distance *MaxAShort*$(p) + 1 +$ *MinATall*$(p) - k + 1 > 2k - 2 - k + 2 = k$ from its witness y, so x is not k-hop dominated by y. Thus, p should necessarily choose its witness outside the subtrees of any of its children (that is, either p declares itself as member of Dom or chooses an ancestor as witness). From (i) and (ii), this means that all nodes in the subtrees of the *tall* children of p adopt a witness different from the one p, and consequently the node x is then the furthest node that belongs to $T(p)$ and has the same witness as p. This implies that $p.\alpha = \|p, x\| =$ *MaxAShort*$(p) + 1$.

- Assume *MaxAShort*(p)+*MinATall*$(p) \leq 2k-2$. Let z be a *tall* child of p such that $z.\alpha =$ *MinATall*(p). Unlike the previous case, p can choose a node y in the subtree of z as witness. Indeed, in this case, x will be at distance *MaxAShort*$(p) + 1 +$ *MinATall*$(p) - k + 1 \leq 2k - 2 - k + 2 = k$ from y. Hence, the nodes (other than p) that are in the subtree of p and has the same witness will be

either nodes in subtrees of short children of p or nodes in $T(z)$. Since by definition, *MinATall*$(p) >$ *MaxAShort*(p), the furthest node that both belongs to $T(p)$ and has the same witness as p will be at distance *MinATall*$(p) + 1$ from p, *i.e.*, $p.\alpha =$ *MinATall*$(p) + 1$.

Proof Scheme. $\mathcal{D}(k)$ is a silent and self-stabilizing algorithm for $\mathbb{P}_{\texttt{kDom}}$. The proof of this result uses Lemma `silent_self_stab` given Section 3 which requires the following two proof obligations:

Partial Correctness: All terminal configurations of $\mathcal{D}(k)$ satisfy $\mathbb{P}_{\texttt{kDom}}$ (`P_correctness`).

Termination: All executions of $\mathcal{D}(k)$ are finite (`termination`).

Proof of `P_correctness` is described in Section 5. Proof of `termination` is described in Section 6.2; it uses general tools for termination that we develop in Section 6.1. Furthermore, we also proved that $|Dom| \leq \lfloor \frac{n-1}{k+1} \rfloor + 1$. The expected property is expressed by:

`counting := n - 1 ≥ (k + 1) * (|Dom| - 1).`

This requirement is the third development of the certified proof:

Counting: All terminal configurations of $\mathcal{D}(k)$ contains at most $\lfloor \frac{n-1}{k+1} \rfloor + 1$ nodes p for which *kDominator*(p) holds.

The way $|Dom|$ and n (number of nodes) are defined in Coq is described in Section 7.1. The proof of `counting` is explained in Section 7.2.

5 Proving Partial Correctness of $\mathcal{D}(k)$

The proof of partial correctness consists in showing that predicate $\mathbb{P}_{\texttt{kDom}}$ holds in any terminal configuration satisfying $\texttt{Assume}_{\texttt{kdom}}$:

Theorem `kdom_set_at_terminal:`
`∀(g: Env), Assume`$_{\texttt{kdom}}$ `g → terminal g →` $\mathbb{P}_{\texttt{kDom}}$ `g.`

Values of α are in range $\{0, ..., 2k\}$. As a preliminary result, we need to check that α, whose domain is `Z` in the Coq definition, matches the range $\{0, ..., 2k\}$. Actually, this might not be true at initial configuration, but this is true both since the very first step and in any terminal configuration. The Coq proof first shows that the value returned by macro $Alpha(p)$ is in range $\{0, ..., 2k\}$: this is proven using a case analysis on *MaxAShort*(n) + *MinATall*$(n) > 2k - 2$ and the fact that, by definition, $-1 \leq$ *MaxAShort*$(n) \leq k - 1$ and $k \leq$ *MinATall*$(n) \leq 2k + 1$. Then, for terminal configuration, the result comes from the fact that $p.alpha = Alpha(p)$.

Tree Paths. From definition of $\mathbb{P}_{\texttt{kDom}}$, we need to check the existence of a path in G between any node `p` and any node `kdom` of Dom, such that this path is of length at most k. To achieve this property, the algorithm builds tree paths of particular shape: those paths use edges of T in both direct sense (from a node to its parent) and reverse sense (from a node to one of its children). Precisely, these edges are defined using relation `is_kDom_edge`, which depends on α-values: for any short node s, we select the edge linking s to its parent in T (using `Par`); while for any tall node t which is not in Dom, we select the edge linking t to a

child c (using `(flip Par)`) such that $c.\alpha = t.\alpha - 1$. The relation `is_kDom_edge` defines a subgraph of G called `kdom`-graph.

To prove Theorem `kdom_set_at_terminal`, it is sufficient to prove that for any configuration `g` where $(\texttt{Assume}_{\texttt{kdom}}\ \texttt{g})$ and `(terminal g)` hold, we have:

```
∀(p: Node), ∃(kdom: Node), (kDominator g kdom) ∧
  ∃(path: list Node), (is_kDom_path g path kdom p) ∧ (length path) ≤ k.
```

where `is_kDom_path` checks that its parameter `path` is a path on the `kdom`-graph between `kdom` and `p`. The rest of the analysis is conducted assuming a terminal configuration `g` which contains a rooted spanning tree built upon a bidirectional graph, namely such that $(\texttt{Assume}_{\texttt{kdom}}\ \texttt{g})$ and `(terminal g)` hold. The proof is split into two cases, depending on whether the node is tall or short.

Proof for Tall Nodes. First, we prove a lemma, called `Alpha_inv`, stating that any node p satisfying $p.\alpha > 0$ has a child q such that $p.\alpha = q.\alpha + 1$. The proof is simply a case analysis on *MaxAShort*(q) + *MinATall*$(q) \leq 2k - 2$. We then prove the following lemma:

Lemma `tall_is_kDominated`:
```
∀(p: Node) (i: nat), (alpha (g p)) = k + Z_of_nat i →
  ∃(kdom: Node), (kDominator g kdom) ∧
  ∃(path: list Node), (is_kDom_path g path kdom p) ∧ (length path) ≤ i.
```

(`Z_of_nat` is a wrapper which transforms a natural number into integer of type `Z`.) The proof straightforward by induction on `i`:

- For `(i = 0)`, p satisfies `kDominator(p)`.
- For `(i = j + 1)`, we apply Lemma `Alpha_inv` to node p such that $p.\alpha$ `= (k + j)`. This provides node `q` with $q.\alpha$ `= (k + i)` on which we apply the induction hypothesis. This exhibits a path `path` of length at most `j` in `kdom`-graph from some k-dominator `kdom` to q. Since p is the parent of q in T, we can build a path `path'` of length at most `j+1 = i` linking `kdom` to p in `kdom`-graph.

Proof for Short Nodes. We now want to prove the similar lemma for short nodes:

Lemma `short_is_kDominated`:
```
∀(p: Node) (i: nat), (alpha (g p)) = k - Z_of_nat i →
  ∃(kdom: Node), (kDominator g kdom) ∧
  ∃(path: list Node), (is_kDom_path g path kdom p) ∧ (length path) ≤ i.
```

We also proceed by induction. The case `(i=0)` is already proved by Lemma `tall_is_kDominated`. For the case `(i = j + 1)`, we look at the tree parent q of p. If q is short, we apply the induction hypothesis, add a step to the path and we are done. If q is tall, we have two cases:

- If *MaxAShort*(q) + *MinATall*$(q) > 2k - 2$, $q.\alpha =$ *MaxAShort*$(q) + 1 \leq k$. Since q is tall, $q.\alpha \geq k$ so $q.\alpha = k$. Since q is the parent of p in the tree, there is a path of length 1 in `kdom`-graph and we are done.
- If *MaxAShort*(q) + *MinATall*$(q) \leq 2k - 2$, we have

  $$(\textit{MinATall}(q) + 1) - k + 1 \leq k - \textit{MaxAShort}(q)$$

  Since p is a short child of q in T, $p.\alpha \leq$ *MaxAShort*(q) or equivalently

  $$k - \textit{MaxAShort}(q) \leq k - p.\alpha$$

We also have, $\mathit{MinATall}(q) + 1 = q.\alpha$. Combining those three equations, we have

$$q.\alpha - k + 1 \leq k - p.\alpha$$

Since q is tall, by `tall_is_kDominated`, there is a node `kdom` and a path `path` of length at most $q.\alpha - k$ in `kdom`-graph from `kdom` to q. Since q is the parent of p in the tree, we have a path `path'` in `kdom`-graph of length at most $q.\alpha - k + 1$ from `kdom` to p. From last inequality, `path'` has a length at most $k - p.\alpha$, so we are done.

Overall Result. Since `g` is a terminal configuration, all the values α are between 0 and $2k$. This result, Lemmas `tall_is_kDominated`, and `short_is_kDominated` imply that all short and tall nodes are within distance k of a node of the k-hop dominating set Dom. Since a node is either short or tall, we have therefore proven Theorem `kdom_set_at_terminal`. As a simple rewriting, `(P_correctness` $\mathbb{P}_{\texttt{kDom}}$`)` is proven.

6 Termination

Termination proofs of self-stabilizing (silent) algorithm often rely on induction scheme, and in particular on *potential functions*. A potential function is a function mapping each configuration to some value (typically an integer) which is monotonic with steps of the algorithm and bounded, *i.e.*, either monotonically decreasing with steps and lower bounded, or monotonically increasing with steps and upper bounded. A potential is usually based on global criteria, *i.e.*, the proof of its monotony usually requires to observe executions at a global level. To simplify the proofs, we developed tools on potential functions and termination at a finer-grained level. We first present these tools and, then, show how to use them to prove termination of $\mathcal{D}(k)$.

6.1 Core Results for Proving Termination

We give a sufficient condition for termination based on some potential function in which the (global) potential function is based on local potential at each node.

Usual termination proofs are based on some global potential built from local ones. For example, local potentials can be integers and the global potential can be the sum of them. In this case, the argument for termination may be, for example, the fact that the global potential is lower bounded and strictly decreases at each step of the algorithm. Global potential decrease is due to the modification of local states at some nodes, however studying aggregators such as sums may hide scenarios, making the proof more complex.

Instead here, we build a global potential as the multiset containing the local potential of each node. Our method for termination is based on two criteria that are sufficient to meet the conditions for using the Dershowitz–Manna well-founded ordering on multisets [DM79]. Given those criteria, we can show that the multiset of (local) potentials globally decreases at each step. For multisets and the Dershowitz-Manna order, we used results from the `CoLoR` library [BK11].

Steps. One difficulty we face, when trying to apply this technique straightly, is that we cannot always define the local potential function at a node without assuming some properties on its local state, and so on the configuration. Thus, we may have to assume the existence of some stable set of configurations in which the local potential function can be defined. When necessary, we use our technique to prove termination of a subrelation of the relation `Step`, provided that the algorithm has been initiated in the required stable set of

configurations. This point is modeled by a predicate on configurations, `safe`, and a type `safeEnv` which represents the set of *safe configurations* into which we restrict the termination proof:

```
safe: Env → Prop
safeEnv: Type := { g: Env | safe g }
```

(`safeEnv` is a type whose values are ordered pairs containing a term `g` and a proof of `(safe g)`.) This set of configurations should be stable, *i.e.*, it is assumed that no step can exit from the set, using the predicate `stable_safe`:

```
stable_safe := ∀ (g g': Env), safe g → Step g' g → safe g'
```

The relation for which termination will be proven is then defined by:

```
safeStep (g' g: safeEnv) := Step (getEnv g') (getEnv g)
```

where `getEnv` accesses the actual configuration (of type `Env`).

Potential. We assume that within safe configurations, each node can be endowed with a potential value obtained using function:

```
pot: safeEnv → Node → Mnat.
```

`Mnat` simply represents natural numbers[8] encoded using the type from Library `CoLoR.MultisetNat` [BK11]; it is equipped with the usual equivalence relation, noted $=_{\mathtt{P}}$, and the usual well-founded order, noted $<_{\mathtt{P}}$, on natural numbers.

Multiset ordering. We recall that a multiset of elements in the setoid, P, endowed with its equivalence relation $=_P$, is defined as a set of elements of P *with a finite number of occurrences with respect to* $=_P$. Such a multiset is usually formally defined as a multiplicity function $m : P \to \mathbb{N}_{\geq 1}$ which maps any element to its number of occurrences in the multiset. We focus here on *finite multisets*, namely, multisets whose multiplicity function has finite support. Now, we assume that P is also ordered using relation $<_P$, compatible with $=_P$. We use the Dershowitz–Manna order on finite multisets [DM79] defined as follows: the multiset N is smaller than the multiset M, noted $N \prec M$, if and only if there are three multisets X, Y and Z such that $X \neq \emptyset \wedge M = Z + X \wedge N = Z + Y \wedge \forall y \in Y, \exists x \in X, y <_P x$, where '+' between multisets means adding multiplicities. Informally, to obtain a multiset N smaller than M, we may remove from M all elements of X and then add all elements of Y. Elements in Z are the ones that are present in both M and N. It is required that some element is removed ($X \neq \emptyset$) and each element that is added must be smaller (*w.r.t.* $<_P$) than some removed element. It has been shown that if $<_P$ is a well-founded order, then so is the corresponding order $\prec$.

In our context, we consider finite multisets over `Mnat`, (*i.e.*, $=_P$ is $=_{\mathtt{P}}$ and $<_P$ stands for $<_{\mathtt{P}}$). We have chosen to model them as lists of elements of `Mnat` and we build the potential of a safe configuration as the multiset of the potentials of all nodes, namely a multiset of potentials of a safe configuration `sg` is defined by

```
Pot (sg: safeEnv): list Mnat := List.map (pot sg) all_nodes
```

where `all_nodes` is the list of every node in the network (see Section 2.2) and `(List.map f l)` is the standard operation that returns the list made of each elements of `l` on which `f` has been applied.

[8] Natural numbers cover many cases and we expect the same results when further extending to other types of potential.

The corresponding Dershowitz–Manna order is defined using the library CoLoR [BK11] with `(≺ := MultisetLT >ₚ)`. The library also contains the proof that `(well_founded <ₚ)` implies `(well_founded ≺)`. Using this latter result and the standard result which proves `(well_founded <ₚ)`, we easily deduce `(well_founded ≺)`.

Termination Theorem. Proving the termination of an algorithm then consists in showing that for any safe step of the algorithm, the corresponding global potential decreases *w.r.t.* the Dershowitz-Manna order $\prec$, namely:

```
safe_incl := ∀ (sg sg': safeEnv), safeStep sg' sg → (Pot sg') ≺ (Pot sg)
```

We established a sufficient condition made of two criteria on node potentials which validate the conditions for using the Dershowitz–Manna well-founded order on multisets $\prec$. *Local criterion* finds for any node p whose potential has changed but has not decreased, a witness node q (typically a neighbor) whose potential has decreased from a value that was even higher than the new potential of p:

```
Hypothesis local_crit:
  ∀ (sg sg': safeEnv), safeStep sg' sg →
    ∀ (p: Node), (pot sg p) <ₚ (pot sg' p) →
      ∃ (q: Node), (pot sg q) ≠ₚ (pot sg' q) ∧ (pot sg' p) <ₚ (pot sg q).
```

Global criterion exhibits, at any step, a node whose potential has changed:

```
Hypothesis global_crit:
  ∀ (sg sg': safeEnv), safeStep sg' sg →
      ∃ (p: Node), (pot sg' p) ≠ₚ (pot sg p).
```

Assuming `local_crit` and `global_crit`, we are able to prove `safe_incl` as follows: we define Z as the multiset of local potentials of nodes whose potential did not change, and X (resp. Y) as the complement of Z in the multiset of local potentials `(Pot sg)` (resp. `(Pot sg')`). Global criterion is used to show that $X \neq \emptyset$, and local criterion is used to show that $\forall y \in Y, \exists x \in X, y <_{\mathtt{p}} x$. Since any relation included in a well-founded order is also well-founded, we get that relation `safeStep` is well-founded.

Lemma `Wf_safeAlgo_Multiset: well_founded safeStep.`

And since we know that property `safe` is stable (from `stable_safe`), we get

```
∀g, safe g → Acc Step g
```

which proves that the algorithm terminates from any safe configuration.

6.2 Proving Termination for $\mathcal{D}(k)$

We use the above method to prove termination of $\mathcal{D}(k)$, expressed as:

```
Theorem k_dom_set_terminates:
  ∀(g: Env), Assume_kdom g → Acc Step g.
```

First, we assume `sym_net` and that the root node *r* exists. We instantiate `safe` as every configuration in which read-only variables `Par` satisfy: `(span_tree r Par)`. Indeed, it is mandatory to assume the spanning tree T rooted at *r* to be able to define local potentials on which we perform the proof since they are based on the depth of nodes in T. Note that it is easy to prove that `safe` is stable since it only depends on read-only variables; `safe` is also proven compatible.

Potential. We define the *depth* of a node as the distance of the node from the root *r* in the tree T. Function `depth` is defined using an induction scheme based on `(WF_par n)`: for a given safe configuration `sg` and node `p`, `(depth sg p)` returns the natural number (type `nat`) defined by 1 if `p` is the root *r* and, otherwise, `1 + (depth sg q)` with `q` the parent of `p` in the tree T, namely such that `(is_tree_parent_of q p)`. We proved that `depth` is a compatible function, always greater than 0, and, when descending a path from the root to some leaf of the tree, depth increases by one at each hop. Furthermore, whatever be the steps computed by the algorithm, depth is preserved. Now, we define the potential of node `p` in safe configuration `sg` as the natural number (type `nat`) defined by:

```
pot sg p := if (enabled_b (getEnv sg) p) then (depth sg p) else 0%nat
```

i.e., the *potential* of node `p` in safe configuration `sg` equals 0 if `p` is not enabled in `sg` and the depth of `p` in the tree T, otherwise. This function is proved compatible.

Local Criterion. Here we fix two safe configurations `sg` and `sg'`, such that `(safeStep sg' sg)`; we note `g` and `g'` the corresponding configurations (obtained using `getEnv`) and we use the proposition `(has_moved p)` for a node `p` which holds when `p` actually executes during step `g` $\mapsto$ `g'` (evaluated by `(d p)` $\neq \bot$ with `d` the difference encoded in the step). We also use predicate `enabled` to represent that `enabled_b` is `true`.

We now detail the proof of local criterion. Let consider a node `p` whose potential has increased during the step, *i.e.*, such that `(pot sg p)` $<_{\mathtt{p}}$ `(pot sg' p)`. This means, from definition of `pot`, that `p` is disabled at `g` (potential is 0) and becomes enabled at `g'` (potential equals `(depth sg' p)` > 0), since depth is preserved during any step. Hence, local criterion is a direct consequence of the following lemma:

Lemma `new_enabled_node_has_disabled_descendant:`

```
∀(p: Node), ¬(enabled g p) → (enabled g' p) →
  ∃(descendant: Node),
    (∃(path: list Node), directed_tree_path g p path descendant) ∧
    (enabled g descendant) ∧ ¬(enabled g' descendant).
```

where `directed_tree_path` means that `path` is a decreasing path along the tree from `p` to `descendant`. Therefore, to prove local criterion, we propose to exhibit a down-path from `p` in T which contains a node, called `descendant`, which is enabled at `g` and becomes disabled at next configuration, `g'`. We prove the lemma in two steps. First, for a node `p`, disabled at `g` but enabled at `g'`, we necessarily exhibit a child of `p`, `child`, which moved during the step:

Lemma `enabled_child_has_moved:`

```
∀(p: Node), ¬(enabled g p) → (enabled g' p) →
  ∃(child: Node), (is_tree_parent_of g p child) ∧ (has_moved child).
```

This is proved by induction on the neighbors of `p` using that the result of `run` only depends on the states of the children of `p` in the tree T (directly based on the definition of `run`, see Algorithm 1).

Hence, as the second step, the proof of Lemma `new_enabled_node_has_disabled_descendant` is reduced to the proof of:

Lemma `moving_node_has_disabled_descendant:`

```
∀(child: Node), has_moved child →
  ∃(descendant: Node),
    (∃(path: list Node), directed_tree_path g child path descendant) ∧
    (enabled g descendant) ∧ ¬(enabled g' descendant).
```

When the node `child` moves, it is down-linked in T to a node which was enabled and becomes disabled, during the step. This result is proven by induction on `(WF_child child)`, *i.e.*, on the decreasing paths from `child` in T. Consider a node in such a path, enabled at `g` and that moves during step `g` $\mapsto$ `g'`: we have two cases.

- Either it becomes disabled at `g'`: this is the basis case of induction, since we have found the witness node `descendant`;
- Or it is still enabled at `g'`: for this case, we prove:

  Lemma `node_has_moved:`
  ```
  ∀(x: Node), has_moved x → enabled g' x →
    ∃(xchild: Node),
      (is_tree_parent_of g x xchild) ∧ (has_moved xchild).
  ```

  (*i.e.*, when a node has moved, but is still enabled at next configuration, it has a child that also moved.) The proof is based on the same schema as Lemma `enabled_child_has_moved`. The lemma provides the induction step for the above proof.

Global Criterion. Global criterion requires to find a witness node whose potential differs between `g` and `g'`. We show that there exists a node `p` with potential `(depth sg p)` > 0 at `g` and potential 0 at `g'`, namely, which is enabled at `g` but disabled at `g'`:

Lemma `one_disabled: ∃(p: Node), (enabled g p) ∧ ¬(enabled g' p).`

Indeed, by definition of the daemon, at least one node has moved during the step. Then Lemma `moving_node_has_disabled_descendant` is used to show that this node necessarily has a descendant (on a given decreasing path of T) which is enabled at `g` but disabled at `g'`.

Conclusion. Local and global criteria being proved, we obtain Theorem `k_dom_set_terminates` directly: this is exactly predicate `termination` for Algorithm $\mathcal{D}(k)$. Using Lemma `silent_self_stab`, we obtain that $\mathcal{D}(k)$ is a silent self-stabilizing algorithm for $\mathbb{P}_{\texttt{kDom}}$:

Theorem `kdom_silent_self_stab:`
`silence` $\texttt{Assume}_{\texttt{kdom}}$ $\wedge$ `self_stab` $\texttt{Assume}_{\texttt{kdom}}$ $\mathbb{S}_{\mathbb{P}_{\texttt{kDom}}}$.

7 Quantitative Properties

In addition to the partial correctness property which states that $\mathcal{D}(k)$ computes a k-hop dominating set Dom, we have shown that $|Dom| \leq \lfloor \frac{n-1}{k+1} \rfloor + 1$, where n the number of nodes. Precisely, we have formally proven the equivalent property `counting` which states that $(n-1) \geq (k+1)(|Dom|-1)$. Intuitively, this means that all but one element of Dom have been chosen as witness by at least $k+1$ distinct nodes each.

7.1 Counting Elements in Sets

First, we have set up a library dealing with cardinality of sets in general and then cardinals of finite sets. The library contains basic properties about set operations such as Cartesian product, disjoint union and subset. Proofs are conducted using standard techniques.

Cardinality on Setoids. To be able to order cardinalities, we define a property, called `Inj`, on a pair of setoids $(A, =_A)$ and $(B, =_B)$: it requires the existence of an injective and compatible function, `inj`, from A to B whose domain is A, namely:

- `Inj_compat`: `inj` is compatible (see Section 2.3),
- `Inj_left_total`: domain of `inj` is A, *i.e.*, any element in A is related to at least one element in B,
- `Inj_left_unique`: `inj` is injective, *i.e.*, any element in B is related to at most one (w.r.t. $=_A$) element in A.

Relation `Inj` is proven reflexive and transitive. We model cardinality ordering using the three-valued type `(Card_Prop := Smaller | Same | Larger)` and the following property `Card`. `Card` distinguishes the different ways `Inj` can apply to pairs of setoids such that

- `(Card Smaller A B)`[9] is defined by `(Inj A B)` which expresses that A has a cardinal smaller or equal to that of B, w.r.t. equalities $=_A$ and $=_B$;
- Similarly, `(Card Larger A B)` is defined by `(Inj B A)`
- and `(Card Same A B)` by `(Inj B A ∧ Inj A B)`.

`(Card prop)` is reflexive and transitive for any value of `prop` in `Card_prop`. It is also antisymmetric in the sense that `(Card Smaller)` and `(Card Larger)` implies `(Card Same)` for a given pair of setoids (trivial from the definitions).

Finite Cardinalities. Now we focus on finite setoids and define tools to express their cardinalities. We first define, for a given natural number N, the setoid $\mathcal{M}_N$ `:= {i: nat | i < N}`. Its values are ordered pairs containing a natural number i and a proof of `(i < N)`; it is equipped with the standard equality on type `nat` (wrapped to be able to compare values of type $\mathcal{M}_N$). In short, $\mathcal{M}_N$ simply models the set of natural numbers $\{0, 1, ..., N-1\}$. We first proved that `Inj` captures finite cardinality ordering

Lemma `Inj_le_iff: ∀(m n: nat), Inj` $\mathcal{M}_m$ $\mathcal{M}_n$ $\leftrightarrow$ $m \leq n$.

and the corresponding corollaries with `Card`, *e.g.*,

`∀(m n: nat), Card Smaller` $\mathcal{M}_m$ $\mathcal{M}_n$ $\leftrightarrow$ $m \leq n$.

(similar corollaries exist for `Larger` and `Same`). The following predicate `Num_Card` is then used to express that a setoid A has cardinality at least (resp. at most, resp. equal to) some natural number n with `Num_Card prop A n := (Card prop A` $\mathcal{M}_n$`)` where `prop` is any `Card_Prop`. For instance, `(Num_Card Smaller A n)` means that A contains at most n elements w.r.t. $=_A$.

Cartesian Products. We developed results about Cartesian products. First, Cartesian product is monotonic w.r.t cardinality:

Lemma `Inj_prod:`
`∀prop, Card prop A1 A2 → Card prop B1 B2 → Card prop (`$A_1 \times B_1$`) (`$A_2 \times B_2$`).`

where $(A_1, =_{A_1})$, $(A_2, =_{A_2})$, $(B_1, =_{B_1})$, $(B_2, =_{B_2})$ are any setoids. Now, we showed that:

[9] We omit parameters $=_A$ and $=_B$ for better readability.

```
∀ n m: nat, Card Same (M_n × M_m) M_{n×m}
```

namely, the Cartesian product of $\mathcal{M}_n = \{0, ..., n-1\}$ and $\mathcal{M}_m = \{0, ..., m-1\}$ contains the same number of elements as $\mathcal{M}_{n\times m} = \{0, ..., n\times m - 1\}$. This latter result is showed using encoding functions from $\mathcal{M}_n \times \mathcal{M}_m$ to $\mathcal{M}_{n\times m}$ and from $\mathcal{M}_{n\times m}$ to $\mathcal{M}_n \times \mathcal{M}_m$. This intermediate result allows to easily deduce that the cardinality of a Cartesian product is the product of cardinalities:

```
∀prop (n m: nat), Num_Card prop A n → Num_Card prop B m →
  Num_Card prop (A × B) (n×m)
```

Disjoint Unions. We developed similar lemmas about disjoint union of sets, noted + for which the main results is:

```
∀prop (n m: nat), Num_Card prop A n → Num_Card prop B m →
  Num_Card prop (A + B) (n + m)
```

Subsets. We proved many toolbox results, about subsets, which are expressed using `Card` as well as `Num_Card`. For instance,

- any subset of a set A has `Smaller` cardinality than that of A,
- the set itself is one of its subset with `Same` cardinality,
- the trivially empty subset contains 0 element,
- a non-empty set contains at least 1 element,
- a singleton contains exactly one element,
- ...

Number of Elements in Lists. To prove the existence of finite cardinality for finite setoids, we use lists, since, in particular, in our framework, the setoid of nodes of the network is encoded as the list `all_nodes`. In this paragraph, we consider a setoid A, whose equality $=_A$ satisfies classical property `(∀a1 a2: A, a1 =_A a2 ∨ a1 ≠_A a2)` and a predicate function `(P: A →` **Prop**`)` which satisfies also classical property `(∀a: A, P a ∨ ¬ P a)`. Under those conditions, we can prove:

```
∀(l: list A), ∃(n: nat), Num_Card Same {a: A | P a ∧ a ∈_{=_A} l} n
```

namely, for any list `l`, the set of elements in `l` (w.r.t. $=_A$) which satisfies predicate `P` has finite cardinality n. Or, equivalently, assuming the existence of a list `l` which contains every element of type A, we get that the number of elements which satisfy `P` is finite:

```
∀(l: list A), (∀(a: A), a ∈_{=_A} l) →
  ∃(n: nat), Num_Card Same {a: A | P a } n
```

When predicate function `P` returns `True` for any parameter, this provides the number of elements of list `l` (up to $=_A$).

7.2 Proving Counting for $\mathcal{D}(k)$

As stated above, we prove that, in a network of n nodes, the final number of nodes in Dom satisfies $(n-1) \geq (k+1)(|Dom| - 1)$. The proof outline is the following. First, we assume a terminal configuration `g` (such that `terminal g`). The existence of the natural number n (number of nodes) is given using the above results about the number of elements in the list `all_nodes`. Similarly, the existence of the natural number

$|Dom|$ (number of nodes in Dom) is given using the above results applied to list `all_nodes` and predicate function `(fun p: Node => (kDominator (g p) = true))`.

We define as *regular head* each node `p` such that α equals k: `RegDomHead p := (α p) = k` and the set of regular heads as `RegDomHeads := { p: Node | RegDomHead (g p) }`. Note that by definition, `RegDomHeads` is included in Dom. Again, we prove the existence of the natural number rh^* which represents the number of nodes in `RegDomHeads` using list `all_nodes` and predicate `RegDomHead`.

We also define a *regular node* as a node which designates a regular head as witness: `RegNodes := { p: Node | HasTallAncestor p }`. Predicate `HasTallAncestor` is an inductive predicate which selects any node which has an ancestor, with α at least k, in the tree T, (namely, there is an increasing path from the node `p` to the root *r* which contains, meanwhile, a node with α at least k). Again, we prove the existence of the natural number rn^* which is the number of nodes in `RegNodes`.

Now, we prove the following theorem:

Theorem `simple_counting:` $rn^* \geq (k+1)rh^*$.

Using results from the library on cardinality of sets and lists, this theorem is reduced to

`Card Smaller (`$\mathcal{M}_{k+1}$ `× RegDomHeads) RegNodes`

This latter proposition is proven by constructing a relation `Rcount` from pairs of natural numbers $i \in \{0, ..., k\}$ and *regular heads* to regular nodes, such that: for a regular head h, some $i \in \{0, ..., k\}$ and a regular node p_i, `(Rcount` (i, h) p_i`)` holds if and only if $p_i.\alpha = i$ and p_i designates h as witness (*i.e.*, there is a path from p_i to h in `kdom`-graph). We show that `Rcount` is actually an injection of domain `(`$\mathcal{M}_{k+1}$ `× RegDomHeads)`. Indeed, for any pair (i, h), there is a node p_i such that $p_i.\alpha = i$ which designates h as witness; the proof is made by induction on values of i. Intuitively, this implies that there is a path of length $k+1$ in `kdom`-graph linking p_0 to h. We then group each regular head with the regular nodes that designate it as witness: each contains at least $k+1$ regular nodes, *i.e*, $rn^* \geq (k+1)rh^*$.

Now, we have two cases. If the root is tall, with $r.\alpha \geq k$, every dominating node (from Dom) is regular (in `RegDomHeads`) and every node is regular (in `RegNodes`). Otherwise, if the root is short, every dominating node is regular but the root and at least one node is not regular, namely the root. These two cases yield the following lemma:

Lemma `split_counting_cases:`
$|Dom| = rh^* \wedge n = rn^* \vee |Dom| = 1 + rh^* \wedge n \geq 1 + rn^*$.

The proof of the lemma first uses the corresponding results on cardinalities (in particular disjoint union between the singleton containing the root and the set of regular nodes (resp. dominating nodes)) and then the above case analysis. The main theorem that proves `Counting` is then just a case analysis from this lemma and proves that $(n-1) \geq (k+1)(|Dom| - 1)$.

8 Conclusion

We proposed a general framework to build certified proofs of self-stabilizing algorithms. To achieve our goals, we, in particular, developed general tools about potential functions, which are commonly used in termination proofs of self-stabilizing algorithms. We also proposed a library dealing with cardinality of sets. We apply this framework to prove that an existing algorithm is silent self-stabilizing for its specification and we show a quantitative property on the output of this case study.

In future works, we expect to certify more complex self-stabilizing algorithms. Such algorithms are usually designed by composing more basic blocks. In this line of thought, we envision to certify general theorems related to classic composition techniques such as collateral or fair compositions.

Finally, we expect to use our experience on quantitative properties to tackle the certification of time complexity of stabilizing algorithms, *aka.* the stabilization time.

References

[ABC+13] Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. Certified Impossibility Results for Byzantine-Tolerant Mobile Robots. In *SSS*, volume 8255, 2013.

[BABB+12] José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full proof cryptography: Verifiable compilation of efficient zero-knowledge protocols. In *ACM Conference on Computer and Communications Security*, pages 488–500, 2012.

[BDPV99] Alain Bui, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Optimal PIF in tree networks. In Yuri Breitbart, Sajal K. Das, Nicola Santoro, and Peter Widmayer, editors, *Distributed Data & Structures 2, Records of the 2nd International Meeting (WDAS 1999), Princeton, USA, May 10-11, 1999*, volume 6 of *Proceedings in Informatics*, pages 1–16. Carleton Scientific, 1999.

[BK11] Frédéric Blanqui and Adam Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011.

[BOBBP13] Jalel Ben-Othman, Karim Bessaoud, Alain Bui, and Laurence Pilard. Self-stabilizing algorithm for efficient topology control in wireless sensor networks. *Journal of Computational Science*, 4(4):199 – 208, 2013. {PEDISWESA} 2011 and Sc. computing for Cog. Sciences.

[CCT13] Eddy Caron, Florent Chuffart, and Cédric Tedeschi. When self-stabilization meets real platforms: An experimental study of a peer-to-peer service discovery system. *Future Generation Computer Systems*, 29(6):1533 – 1543, 2013.

[CDDL10] Eddy Caron, Ajoy Kumar Datta, Benjamin Depardon, and Lawrence L. Larmore. A self-stabilizing k-clustering algorithm for weighted graphs. *J. Parallel Distrib. Comput.*, 70(11):1159–1173, 2010.

[CDL11] Pierre Corbineau, Mathilde Duclos, and Yassine Lakhnech. Certified security proofs of cryptographic protocols in the computational model: An application to intrusion resilience. In *CPP*, pages 378–393, 2011.

[CDPT10] Eddy Caron, Frédéric Desprez, Franck Petit, and Cédric Tedeschi. Snap-stabilizing prefix tree for peer-to-peer systems. *Parallel Processing Letters*, 20(1):15–30, 2010.

[CFG92] Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In *Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 9-12, 1992*, pages 486–493. IEEE Computer Society, 1992.

[CFM09] Pierre Castéran, Vincent Filou, and Mohamed Mosbah. Certifying distributed algorithms by embedding local computation systems in the coq proof assistant. In *Symbolic Computation in Software Science (SCSS'09)*, 2009.

[CM12] Meixian Chen and Jean-François Monin. Formal Verification of Netlog Protocols. In Tiziana Margaria, Zongyan Qiu, and Hongli Yang, editors, *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, pages 43–50. IEEE, 2012.

[Cou02] P. Courtieu. Proving Self-Stabilization with a Proof Assistant. In *IPDPS*. IEEE Computer Society, 2002.

[CRTU15] Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Impossibility of gathering, a certification. *Inf. Process. Lett.*, 115(3):447–452, 2015.

[CYH91] NS Chen, HP Yu, and ST Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[DGS96] S. Dolev, M. G. Gouda, and M. Schneider. Memory Requirements for Silent Stabilization. In *PODC*, pages 27–34, 1996.

[Dij74] E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17:643–644, 1974.

[DLD+12] Ajoy Kumar Datta, Lawrence L. Larmore, Stéphane Devismes, Karel Heurtefeux, and Yvan Rivierre. Competitive self-stabilizing k-clustering. In *ICDCS, 2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012*, pages 476–485. IEEE, 2012.

[DLD+13] Ajoy Kumar Datta, Lawrence L. Larmore, Stéphane Devismes, Karel Heurtefeux, and Yvan Rivierre. Self-stabilizing small k-dominating sets. *IJNC*, 3(1):116–136, 2013.

[DM79] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.

[DM09] Yuxin Deng and Jean-François Monin. Verifying Self-stabilizing Population Protocols with Coq. In *TASE*, pages 201–208, 2009.

[Dol97] Shlomi Dolev. Self-stabilizing routing and related protocols. *Journal of Parallel and Distributed Computing*, 42(2):122 – 127, 1997.

[FMP13] Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *Static Analysis Symposium (SAS)*, volume 7935 of *Lecture Notes in Computer Science*, pages 345–365. Springer, 2013.

[Ga13] G. Gonthier and al. A Machine-Checked Proof of the Odd Order Theorem. In *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, 2013.

[Gho93] Sukumar Ghosh. An alternative solution to a problem on self-stabilization. *ACM Trans. Program. Lang. Syst.*, 15(4):735–742, 1993.

[HC93] Shing-Tsaan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993.

[KNR12] Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms. In JosC.M. Baeten, Tom Ball, and FrankS. de Boer, editors, *Theoretical Computer Science*, volume 7604 of *Lecture Notes in Computer Science*, pages 209–224. Springer Berlin Heidelberg, 2012.

[KRS99] Sandeep S. Kulkarni, John M. Rushby, and Natarajan Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In *WSS*, pages 33–40, 1999.

[Lam12] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, 2012.

[Ler09] Xavier Leroy. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[MP67] John Mccarthy and James Painter. Correctness of a Compiler for Arithmetic Expressions. In *Applied Mathematica*, volume 19 of *Mathematical Aspects of Computer Science*, pages 33–41, 1967.

[STW+13] Gerry Siegemund, Volker Turau, Christoph Weyer, Stefan Lobs, and Jörg Nolte. Brief announcement: Agile and stable neighborhood protocol for wsns. In *Proceedings of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'13)*, pages 376–378, November 2013.

[The12] The Coq Development Team. *The Coq Proof Assistant Documentation*, June 2012.